Single Assignment C: HP^2 programming for heterogeneous concurrent architectures

Scholz, S.-B.; Herhut, S.; Grelck, C.

Publication date
2010
Document Version
Author accepted manuscript
Published in
Intel European Research and Innovation Conference (ERIC'10), Braunschweig, Germany

Single Assignment C: HP^2 Programming for Heterogeneous Concurrent Architectures♦

Sven-Bodo Scholz[1], Stephan Herhut[1], Clemens Grelck[2]

[1] University of Hertfordshire, Hatfield, United Kingdom
[2] University of Amsterdam, Amsterdam, The Netherlands

s.scholz@herts.ac.uk

Summary

The range of architectures used in high-performance computing is quickly expanding and at the same time lifetimes of platforms are decreasing. This shift threatens the return on investment for tuning applications to specific architectures and platforms, which until now was the prevailing development model in high-performance computing. As a result, the reusability of software components and the ability to execute existing code efficiently on existing and future architectures come to be key concerns in high-performance computing. This observation lies at the heart of HP^2 programming: to *combine high-performance with high-productivity*. With Single Assignment C and its attending compiler technology, we have developed a programming system that puts HP^2 to practice.

1 Single Assignment C

Single Assignment C [1], or SAC for short, is a programming languages geared towards efficient specification and execution of numerical applications on concurrent hardware. It was designed from ground up with *programmer productivity* and *runtime efficiency* in mind.

To facilitate the former, SAC foots on the familiar syntax of C, extended by an array calculus similar to that of APL and MATLAB. Memory management and data placement is fully implicit in SAC to allow the programmer to concentrate on the task at hand.

Achieving the latter is the result of more than a decade in compiler research. Array computations in SAC ultimately boil down to a single, potentially parallel operation similar to the well-known map and reduce operations. That operation exposes the application's implicit concurrency at a fine granular level. Using a variety of transformations, sequences of such operation are mapped efficiently to platforms of varying granularity, from sequential execution on a single core to thousands of threads on experimental hardware.

Our compiler suite translates SAC to a variety of targets. Combined with a flexible foreign language interface, this makes SAC ideally suited as a domain specific language for computationally intensive kernels within larger applications.

2 Evaluation

SAC has been evaluated in a range of projects to assess its suitability to quickly port existing applications, or hot kernels thereof, that were written in industry standard imperative languages like C or Fortran to a range of platforms. A. Shafarenko et al. have shown that an experienced Fortran programmer can quickly become productive in SAC. The resulting portable code even outperforms its Fortran counterpart [2]. Similarly, A. Kudryavtsev et al. report that they observed competitive sequential runtimes and, more importantly, automatic scaling on SMP machines [3]. In the context of OpenModelica, K. Stavåker, et al. have used SAC as a backend for their own compiler, thereby unlocking the power of GPGPU for their tool chain with little investment [4].

Publications

[1] C. Grelck and S.-B. Scholz. SAC: A Functional Array Language for Efficient Multithreaded Execution. Int. J. Parallel Prog, 34(4):383–427, 2006.

[2] A. Shafarenko, et al. Implementing a Numerical Solution of the KPI Equation using Single Assignment C: Lessons and Experiences. In: IFL'05. Selected Papers. LNCS 4015. Springer, 2006.

[3] A. Kudryavtsev, et al. Numerical Simulations of Unsteady Shock Wave Interactions using SAC and Fortran-90. In: PaCT'09. LNCS 5698. Springer, 2009.

[4] K. Stavåker, et al. Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPU. In: EOOLT'10, Oslo, Norway, 2010.

♦ Supported by research grants FP7/2007/215216 Apple-CORE and FP7/2009/248828 ADVANCE.